Certified In-lined Reference Monitoring on .NET [*]

Kevin W. Hamlen
Cornell University
hamlen@cs.cornell.edu

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

Fred B. Schneider
Cornell University
fbs@cs.cornell.edu

Abstract

Mobile is an extension of the .NET Common Intermediate Language that supports certified In-Lined Reference Monitoring. Mobile programs have the useful property that if they are well-typed with respect to a declared security policy, then they are guaranteed not to violate that security policy when executed. Thus, when an In-Lined Reference Monitor (IRM) is expressed in Mobile, it can be certified by a simple type-checker to eliminate the need to trust the producer of the IRM.

Security policies in Mobile are declarative, can involve unbounded collections of objects allocated at runtime, and can regard infinite-length histories of security events exhibited by those objects. The prototype Mobile implementation enforces properties expressed by finite-state security automata—one automaton for each security-relevant object—and can type-check Mobile programs in the presence of exceptions, finalizers, concurrency, and non-termination. Executing Mobile programs requires no change to existing .NET virtual machine implementations, since Mobile programs consist of normal managed CIL code with extra typing annotations stored in .NET attributes.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.2.1 [*Software Engineering*]: Requirements/Specifications; D.4.6 [*Operating Systems*]: Security and Protection—Access controls; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

General Terms Security

Keywords program rewriting, reference monitors, execution monitoring, in-lined reference monitoring, security automata

[*] Supported in part by AFOSR grant F49620-03-1-0156, National Science Foundation Grants 0430161 and CCF-0424422 (TRUST), ONR Grant N00014-01-1-0968, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

PLAS'06 June 10, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-374-3/06/0006…$5.00.

1. Introduction

Language-based approaches to computer security have employed two major strategies for enforcing security policies over untrusted programs.

- Low-level type systems, such as those used in Java bytecode [22], .NET CIL [10], and TAL for x86 [24], can enforce important program invariants such as *memory safety* and *control safety*, which dictate that programs must access and transfer control only to certain suitable memory addresses throughout their executions. Proof-Carrying Code (PCC) [25] generalizes the type-safety approach by providing an explicit proof of safety in first-order logic.
- *Execution Monitoring* technologies such as Java and .NET stack inspection [15] [22, II.22.11], SASI [12], Java-MAC [21], Java-MOP [4], Polymer [1], and Naccio [13], use runtime checks to enforce temporal properties that can depend on the history of the program's execution. For example, SASI Java was used to enforce the policy that no program may access the network after it reads from a file [11]. For efficiency, execution monitors are often implemented as *In-lined Reference Monitors (IRM's)* [28], wherein the runtime checks are in-lined into the untrusted program itself to produce a *self-monitoring program*.

The IRM approach is capable of enforcing a large class of powerful security policies, including ones that cannot be enforced with purely static type-checking [18]. In addition, IRM's can enforce a flexible range of policies, often allowing the code recipient to choose the security policy after the code is received, whereas static type systems and PCC usually enforce fixed security policies that are encoded into the type system or proof logic itself, and that therefore cannot be changed without changing the type system or certifying compiler.

But despite their power and flexibility, the *rewriters* that automatically embed IRM's into untrusted programs are typically trusted components of the system. Since rewriters tend to be large and complex when efficient rewriting is required or complex security policies are to be enforced, the rewriter becomes a significant addition to the system's trusted computing base.

In this paper, we present Mobile, an extension to the .NET CIL that makes it possible to automatically verify IRM's using a static type-checker. Mobile (MOnitorable BIL with Effects) is an extension of BIL (Baby Intermediate Language) [16], a substantial fragment of managed .NET CIL that was used to develop generics for .NET [20]. Mobile programs are CIL programs with additional typing annotations that track an abstract representation of program execution history. These typing annotations allow a type-checker to verify statically that the runtime checks in-lined into the untrusted program suffice to enforce a specified security policy. Once type-checked, the typing annotations can be erased, and the self-monitoring program can be safely executed as normal CIL code. This verification process allows a rewriter to be removed from the

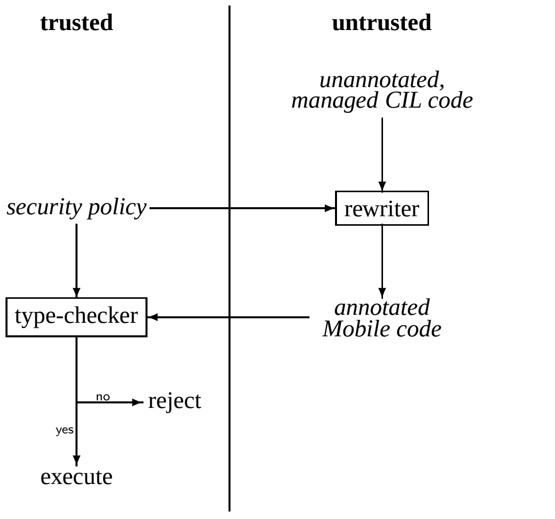

Figure 1. A Mobile load path

trusted computing base and replaced with a (simpler) type-checker. Even when the rewriter is small and therefore comparable in size to the type-checker, type-checking constitutes a useful level of redundancy that provides greater assurance than trusting the rewriter alone. Mobile thus leverages the power of IRM's while using the type-safety approach to keep the trusted computing base small.

Figure 1 summarizes a typical load path on a system that executes IRM's written in Mobile. Untrusted, managed CIL code is first automatically rewritten according to a security policy, yielding an annotated, self-monitoring program written in Mobile. The rewriting can be performed by either a code producer or by a client machine receiving the untrusted code. Since the rewriter, and therefore the self-monitoring program, remains untrusted, the self-monitoring program is then passed to a trusted type-checker that certifies the code with respect to the original security policy. Code that satisfies the security policy will be approved by the type-checker, and is therefore safe to execute; code that is not well-typed will be rejected and would indicate a failure of the rewriter.

In this paper we focus on robust certification of Mobile code. Techniques for efficient rewriting are left to future work, but we describe a naïve rewriter and suggest some strategies for optimizing it in §3. Our prototype implementation of Mobile consists of a type-checker that verifies sound rewriting with respect to security policies expressed as ω-regular expressions. The implementation can verify both single-threaded and multi-threaded managed CIL applications, and it supports language features beyond those modeled by BIL, such as exceptions and finalizers.

2. Related Work

Type-systems $\lambda_{\mathcal{A}}$ [32] and λ_{hist} [29] enforce history-based security policies over languages based on the λ-calculus. In both, program histories are tracked at the type-level using effect types that represent an abstraction of those global histories that might have been exhibited by the program prior to control reaching any given program point.

Mobile differs from $\lambda_{\mathcal{A}}$ and λ_{hist} by tracking history on a per-object basis. That is, both $\lambda_{\mathcal{A}}$ and λ_{hist} represent a program's history as a finite or infinite sequence of global program events, where the set of all possible global program events is always finite. Policies that are only expressible using an infinite set of global program events (e.g., events parameterized by object instances) are therefore not enforceable by $\lambda_{\mathcal{A}}$ or λ_{hist}. For example, the policy that every opened file must be closed by the time the program terminates is not enforceable by either $\lambda_{\mathcal{A}}$ or λ_{hist} when the number of file objects that could be allocated during the program's execution is unbounded. In object-oriented languages such as the .NET CIL, policies concerning unbounded collections of objects arise naturally, so it is not clear how $\lambda_{\mathcal{A}}$ or λ_{hist} can be extended to such settings. Mobile enforces policies that are universally quantified over objects of any given class, and therefore allows objects to be treated as first-class in policy specifications.

PCC has been proposed as a framework for supporting certifying rewriting using temporal logic [3]. The approach is potentially powerful, but does not presently support languages that include exceptions, concurrency, and other features found in real programming languages [2, p. 173]. It is therefore unclear whether proof size and verification speed would scale well in practical settings.

CQual [14] and Vault [7] are C-like languages that enforce history-based properties of objects by employing a flow-sensitive type system based on alias types [30] and typestates [9]. Security-relevant objects in CQual or Vault programs have their base types augmented with type qualifiers, which statically track the security-relevant state of the object. A type-checker then determines if any object might enter a state at runtime that violates the security policy. Vault's type system additionally includes variant types that allow a runtime value to reflect an object's current state. The Vault type-checker identifies instructions that test these state values to ensure that those tests will prevent security violations when the program is executed.

Fugue [8] is a static verifier based on Vault that uses programmer-supplied specifications to find bugs in .NET source code. It verifies policies that constrain the use of system resources or that prescribe protocols that constrain the order in which methods may be called on objects. Fugue supports any source language that compiles to managed .NET CIL code, but it does not support exceptions, finalizers, or concurrency. It additionally lacks a formal proof of soundness for its aliasing analysis and type system.

Inspired by CQual, Vault, and Fugue, our work scales these ideas up to a large existing programming language—the full managed .NET CIL (minus reflection)—while providing a formal proof of soundness. In scaling up to a larger-scale language, we adopt a somewhat different approach to tracking object security states at the type level. CQual, Vault, and Fugue assign linear types to security-relevant objects (and, in the case of Vault, to runtime state values), and use aliasing analyses to track changes to items with linear types. However, it is not clear how such analyses can be extended to support concurrency or to support an important technique commonly used by IRM's to track object security states, wherein security-relevant objects are paired with runtime values that record their states, and then such pairs are permitted to leak to the heap. Existing alias analyses cannot easily track items that are permitted to leak to the heap arbitrarily, or that are shared between threads.

We therefore take the approach of L^3 [23], wherein linearly-typed items are permitted to leak to the heap by packing them into shared data structures with limited interfaces. These shared object-state pairs, called *packages*, can be aliased arbitrarily and are not tracked by the type system. Mobile provides trusted operations for packing and unpacking linear-typed items to and from shared package objects. To perform any (security-relevant) operation that might change a value with linear type, it must first be unpacked from any package that contains it. As with ownership types [6, 5], packing and unpacking operations are implemented as destructive reads, so that only one thread can perform security-relevant operations on a given security-relevant object at a time. Mobile's type system and the CLI permissions system are leveraged to maintain invariants linking an object to an accurate runtime representation of its state.

3. Overview

A Mobile *security policy* identifies a set of security-relevant object classes and assigns a set of acceptable *traces* to each such class. A trace is a finite or infinite sequence of security-relevant *events*—program operations that take a security-relevant object as an argument. Our implementation expresses security policies as ω-regular expressions over the alphabet of events, but the formalisms presented in this article can be leveraged to support alternative policy languages as well. A Mobile program *satisfies* the security policy if for every complete run of the program, (i) if the run is finite (i.e., the program terminates), the sequence of security-relevant events performed on every object allocated during that run is a member of the set of traces that the security policy has assigned to that object's class; and (ii) if the run is infinite (i.e., the program does not terminate), at each step of the run the sequence of security-relevant events performed so far on each security-relevant object is a prefix of a member of the set of traces assigned to that object's class.

For example, [8, p. 5] proposes a security policy involving a `WebPageFetcher` class for which proper usage is to call the `Open` method to acquire the resource, the `GetPage` method to use the resource, and the `Close` method to release the resource. A Mobile policy that requires programs to open web pages before reading them, allows at most three reads per opened page, and requires programs to close web pages before the program terminates (but allows them to remain open on runs that never terminate), might assign $(\mathtt{O}\,(\mathtt{G} \cup \mathtt{G}^2 \cup \mathtt{G}^3)\,\mathtt{C})^\omega$ as the set of acceptable traces for class `WebPageFetcher` (where `O`, `G`, and `C` denote `Open`, `GetPage`, and `Close` events, respectively, and ω denotes finite or infinite repetition). Note that this policy can only be enforced by a mechanism that tracks events at a per-object level.

Although Mobile security policies model events as operations performed on objects, *global events* that do not concern any particular object can be encoded as operations on a *global object* that is allocated at program start and destroyed at program termination. Thus, Mobile policies can regard global events, per-object events, and combinations of the two.

For example, one might modify the example policy above by additionally requiring that at most ten network sends may occur during the lifetime of the program. In that case, the global object would additionally be identified as a security-relevant object, a `Send` method call performed on any `System.Net.Sockets.Socket` object would be identified as a security-relevant event for the global object, and the global object would be assigned the set of traces denoted by $\epsilon \cup \mathtt{S} \cup \mathtt{S}^2 \cup \cdots \cup \mathtt{S}^{10}$ (where `S` denotes a `Send` event).

A rewriter that produces self-monitoring programs from untrusted CIL code is expected to produce well-typed Mobile code, so that the policy-adherence theorem can be used to guarantee that it is safe to execute. For this rewriting task to be feasible, Mobile's type system must be flexible enough to permit rewriters to insert runtime security checks—well-typed code that tracks the state of security-relevant objects at runtime, testing aspects of the state that cannot be verified statically. To that end, Mobile supports a **pack** operation that pairs a security-relevant object with a runtime value (e.g., an integer) representing an abstraction of the object's current state, and that encapsulates them into a two-field package object. Mobile's **unpack** operation can be used to unpack a package, yielding the original object that was packed along with the runtime value that represents its state. Mobile programs can then test this runtime value to infer information about the associated object's state. Both **pack** and **unpack** are implemented as CIL method calls to a small trusted library (about ten lines of C# code).

To keep type-checking tractable, Mobile does not allow security-relevant operations on objects that are packed. A package class's two fields are declared to be `private` so that, to access a security-relevant object directly and perform operations on it, it must first be unpacked. While unpacked, Mobile allows only limited aliasing of security-relevant objects—none of their aliases can escape to the heap. To enforce this restriction, the **unpack** operation is implemented as a destructive read, preventing the package from being unpacked again before it is re-packed. Packages, however, are permitted to escape to the heap and to undergo unlimited aliasing. These restrictions allow the type-checker to statically track histories of unpacked objects and to ensure that packed objects are always paired with a value that accurately reflects their state. When an object is packed, it is safe for the type-checker to forget whatever information might be statically known about the object, keeping the type-checking algorithm tractable and affording the rewriter a dynamic fallback mechanism when static analysis cannot verify all security-relevant operations.

When **pack** and **unpack** are implemented as atomic operations, Mobile can also enforce security policies in concurrent settings. In such a setting, Mobile's type system maintains the invariant that each security-relevant object is either packed or held by at most one thread. Packed objects are always policy-adherent (or their finalizers must bring them to a policy-adherent state at program termination; see §5), whereas unpacked objects are tracked by the type system to ensure that they return to a policy-adherent state before they are relinquished by the thread.

Implementing **pack** and **unpack** as atomic swaps is a somewhat blunt approach, but it is still powerful enough to support useful and effective rewriting strategies. Using the above operations, a naïve rewriter can implement state-based histories by simply representing security-relevant objects as packages. Whenever a security-relevant operation is to be performed, the rewriter would insert code to first unpack the package and test the object's runtime state, then perform the security-relevant operation only if the test succeeds (possibly terminating otherwise), and finally repackage the object with updated state.

This strategy suffices to implement any state-based history but might result in inefficient code if security-relevant operations are frequent. Thus, Mobile's type system also makes it possible to avoid some of these dynamic operations when policy-adherence can be proved statically. For example, a more sophisticated rewriter could in some cases insert code to perform numerous security-relevant operations consecutively without any dynamic checks. Instead of dynamic checks, the rewriter could add typing annotations that prove to the type-checker that the omitted checks are unnecessary for preventing a security violation. Substituting annotations for dynamic checks in this way is often possible in straight-line code or tight loops that do not leak security-relevant objects to the heap. However, when objects do escape to the heap, the type system is not sufficiently powerful to track them and dynamic checks would usually be necessary in order to prove that a security violation cannot occur. Thus, Mobile's type system is sufficiently expressive that rewriters can avoid some but not all dynamic checks.

Our implementation of Mobile models security policies as finite-state security automata. This approach is appealing because it is simple, practical, it introduces minimal extra state to untrusted programs, and it seems to cover most of the enforceable security policies discussed in the literature. However, the formalisms presented in this paper do not assume any particular method of representing object states at runtime. Rather, we parameterize the framework in terms of arbitrary state representations and state tests so that alternative implementations can be realized in the future. For example, future implementations might track object states using LTL expressions or even by recording an object's complete history at runtime. Thus, Mobile constitutes a framework general enough to reason about many different in-lining strategies used by IRM's.

$I ::= \mathbf{ldc.i4}\ n$	integer constant
$I_1\ I_2\ I_3\ \mathbf{cond}$	conditional
$I_1\ I_2\ \mathbf{while}$	while-loop
$I_1; I_2$	sequence
$\mathbf{ldarg}\ n$	method argument
$I\ \mathbf{starg}\ n$	store into arg
$I_1\ \dots\ I_n\ \mathbf{newobj}\ C(\mu_1, \dots, \mu_n)$	make new obj
$I_0\ I_1\ \dots\ I_n\ \mathbf{callvirt}\ C::m.Sig$	method call
$I\ \mathbf{ldfld}\ \mu\ C::f$	load from field
$I_1\ I_2\ \mathbf{stfld}\ \mu\ C::f$	store into field
$I\ \mathbf{evt}\ e$	exhibit event
$\mathbf{newpackage}\ C$	make new package
$I_1\ I_2\ I_3\ \mathbf{pack}$	pack package
$I\ \mathbf{unpack}\ n$	unpack package
$I_1\ I_2\ I_3\ \mathbf{condst}\ C, k$	test state
$I_1\ \dots\ I_n\ \mathbf{newhist}\ C, k$	state constructor
$\boxed{v}$	values
$I\ \mathbf{ret}$	method return

Figure 2. The Mobile instruction set

4. A Formal Analysis of Mobile

4.1 The Abstract Machine

Figure 2 gives the Mobile instruction set. Like BIL, Mobile's syntax is written in postfix notation. In addition to BIL instructions,[1] Mobile includes

- instruction $\mathbf{evt}\ e$, which performs security-relevant operation e on an object (where e is some unique identifier, such as "open", that we associate with each security-relevant operation),
- instructions $\mathbf{newpackage}$ and $\mathbf{newhist}$ for creating packages and runtime state values,
- instructions $\mathbf{pack}$ and $\mathbf{unpack}$ for packing/unpacking objects and runtime state values to/from packages,
- instruction $\mathbf{condst}$, which dynamically tests a runtime state value, and
- the pseudo-instructions $\boxed{v}$ and $\mathbf{ret}$, which do not appear in source code but are introduced in the intermediate stages of the small-step semantics presented in §4.2. (Instruction $\boxed{v}$ is a term that has been reduced to value v, and instruction $\mathbf{ret}$ pops the current stack frame at the end of a method call.)

These abstract instructions model real CIL instructions. For example, if calls to method m are security-relevant operations, the CIL instruction that invokes m on object o is modeled by the Mobile instruction sequence: $o\ \mathbf{evt}\ e_m; o\ \mathbf{callvirt}\ C::m.Sig_m$. A description of how our implementation models other CIL instructions is given in §5.

Figure 3 provides Mobile's type system. Mobile types consist of void types, integers, classes, and *history abstractions* (the types of runtime state values). The type of each unpacked, security-relevant object $C\langle\ell\rangle$ is parameterized by an *object identity variable* ℓ that uniquely identifies the object. All aliases of the object have types with the same object identity variable, but other unpacked objects of the same class have types with different object identity

[1] For simplicity, we omit BIL's value classes and managed pointers from Mobile, but otherwise include all BIL types and instructions.

Types	$\tau ::= \mu \mid C\langle\ell\rangle$
Untracked types	$\mu ::= \mathbf{void} \mid \mathbf{int32} \mid C\langle ? \rangle \mid \mathit{Rep}_C\langle H\rangle$
Class names	C
Object identity variables	ℓ
History abstractions	$H ::= \epsilon \mid e \mid H_1 H_2 \mid H_1 \cup H_2 \mid H^\omega \mid \theta \mid H_1 \cap H_2$
History abstraction variables	θ
Method signatures	$Sig ::= \forall\Gamma_{in}.((\Psi_{in}, Fr_{in}) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau))$
Typing contexts	$\Gamma ::= \cdot \mid \Gamma,\ \ell{:}C \mid \Gamma,\ \ell{:}C\langle ? \rangle \mid \Gamma, \theta$
Object history maps	$\Psi ::= 1 \mid \Psi \star (\ell \mapsto H)$
Local variable frames	$Fr ::= (\tau_0, \dots, \tau_n)$

Figure 3. The Mobile type system

$$\frac{}{\tau \preceq \tau}$$

$$\frac{H \subseteq H'}{\mathit{Rep}_C\langle H\rangle \preceq \mathit{Rep}_C\langle H'\rangle}$$

$$\frac{\tau_i \preceq \tau_i' \quad \forall i \in 0..n}{(\tau_0, \dots, \tau_n) \preceq (\tau_0', \dots, \tau_n')}$$

$$\frac{\mathit{Dom}(\Psi) = \mathit{Dom}(\Psi') \qquad \Psi(\ell) \subseteq \Psi'(\ell)\ \forall \ell \in \mathit{Dom}(\Psi)}{\Psi \preceq \Psi'}$$

Figure 4. Mobile subtyping

variables. The types $C\langle ? \rangle$ of packed classes and security-irrelevant classes do not include object identity variables, and their instances are therefore not distinguishable by the type system. We consider Mobile terms to be equivalent up to alpha conversion of bound variables.

The types $\mathit{Rep}_C\langle H\rangle$ of runtime state values are parameterized both by the class type C of the object to which they refer and by a *history abstraction* H—an ω-regular expression (plus variables and intersection) that denotes a set of traces. In such an expression, ω denotes finite or infinite repetition.

Closed (i.e., variable-less) history abstractions conform to a subset relation; we write $H_1 \subseteq H_2$ if the set of traces denoted by H_1 is a subset of the set of traces denoted by H_2. This subset relation induces a natural subtyping relation $\preceq$ given in Figure 4. Observe that the subtyping relation in Figure 4 does not recognize class subtyping of security-relevant classes. We leave support for subtyping of security-relevant classes to future work.

Type variables in Mobile types are bound by typing contexts Γ, which assign class or package types to object identity variables ℓ and declare any history abstraction variables θ. Object identity variables can additionally appear in object history maps Ψ, which associate a history abstraction H with each object identity variable that corresponds to an unpacked, security-relevant object. Since object identity variables uniquely identify each object instance, object history maps can be seen as a spatial conjunction ($\star$) [26] of assertions about the histories of the various unpacked objects in the heap.

A complete Mobile program consists of:

Class names	C
Field types	$\mathit{field} : (C \times f) \to \mu$
Class methods	$\mathit{methodbody} : (C::m.Sig) \to I$
Class policies	$\mathit{policy} : C \to H$

$v ::=$	result
$\mathbf{0}$	void
$i4$	integer
ℓ	heap pointer
$\mathit{rep}_C(H)$	runtime state value
$o ::=$	heap elements
$\mathit{obj}_C\{f_i = v_i\}^{\overrightarrow{e}}$	object
$\mathit{pkg}(\ell, \mathit{rep}_C(H))$	filled package
$\mathit{pkg}(\cdot)$	empty package
$h ::= \ell_i \mapsto o_i$	heap
$a ::= (v_0, \ldots, v_n)$	arguments
$s ::= (a_0, \ldots, a_n)$	stack
$\psi ::= (h, s)$	small-step store

Figure 5. The Mobile memory model

We also use the notation $\mathit{fields}(C)$ to refer to the number of fields in class C. Method signatures Sig will be described in §4.3.

4.2 Operational Semantics

Unlike [16], we provide a small-step operational semantics for Mobile rather than a large-step semantics, so as to apply the policy adherence theorems presented in §4.4 to programs that do not terminate or that enter a bad state.

In Mobile's small-step memory model, presented in Figure 5, objects consist not only of an assignment of values to fields but also a trace $\overrightarrow{e}$ that records a history of the security-relevant operations performed on the object. Although our model attaches a history trace to each object, we prove in §4.4 that it is unnecessary for the virtual machine to track and store object traces because well-typed Mobile code never exhibits a trace that violates the security policy.

The small-step operational semantics of Mobile, given in Figures 6 and 7, define how a given store ψ and instruction I steps to a new store ψ' and instruction I', written $\psi, I \leadsto \psi', I'$. Rules 13–18 model the behavior of the new instructions introduced by Mobile. Rule 13 appends event e_1 to the sequence of events exhibited on object ℓ. Rule 14 introduces a new package object to the local context. Rule 15 assigns an object ℓ' and runtime state value $\mathit{rep}_C(H)$ to the fields of package ℓ. Rule 16 yields the object and runtime state value stored in package ℓ and erases ℓ's fields.

Rules 17 and 18 use notation not previously defined and therefore deserve special note. Runtime operations $\mathrm{test}_{C,k}$ and $\mathrm{hc}_{C,k}$ test runtime state values and construct new runtime state values, respectively. Rather than fixing these two operations, we allow Mobile to be extended with unspecified implementations of them. Different implementations of $\mathrm{test}_{C,k}$ and $\mathrm{hc}_{C,k}$ can therefore be used to allow Mobile to support different collections of security policies. For example, a Mobile system that supports security policies expressed as DFA's might implement runtime state values as 32-bit integers and might support tests that compare runtime state values to integer constants (to determine which state the DFA is in). In that case, one could define for each $k \in 0..2^{32}$, $\mathrm{hc}_{C,k}() = k$ and $\mathrm{test}_{C,k}(i) = \{1 \text{ if } i = k, \text{ else } 0\}$. A more powerful (but more computationally expensive) Mobile system might implement runtime state values as dynamic data structures that record an object's entire trace and might provide tests to examine such structures. In this paper, we assume only that a countable collection of state value constructors and tests exists and that this collection adheres to typing constraints 19, 20, 21, and 22 presented in §4.3.

$$\begin{aligned}
E ::= & [\,] \mid E\ I_2\ I_3\ \mathbf{cond} \mid E; I_2 \mid E\ \mathbf{starg}\ n \mid \\
& \boxed{v_1}\ \ldots\ \boxed{v_m}\ E\ I_1\ \ldots\ I_n\ \mathbf{newobj}\ C(\mu_1, \ldots, \mu_{m+n+1}) \mid \\
& \boxed{v_1}\ \ldots\ \boxed{v_m}\ E\ I_1\ \ldots\ I_n\ \mathbf{callvirt}\ C{::}m.\mathit{Sig} \mid E\ \mathbf{ret} \mid \\
& E\ \mathbf{ldfld}\ \mu\ C{::}f \mid E\ I_2\ \mathbf{stfld}\ \mu\ C{::}f \mid \boxed{v_1}\ E\ \mathbf{stfld}\ \mu\ C{::}f \mid \\
& E\ \mathbf{evt}\ e \mid E\ I_2\ I_3\ \mathbf{pack} \mid \boxed{v_1}\ E\ I_3\ \mathbf{pack} \mid \boxed{v_1}\ \boxed{v_2}\ E\ \mathbf{pack} \mid \\
& E\ \mathbf{unpack}\ C, k \mid E\ I_2\ I_3\ \mathbf{condst}\ C, k \mid \\
& \boxed{v_1}\ \ldots\ \boxed{v_m}\ E\ I_1\ \ldots\ I_n\ \mathbf{newhist}\ C, k
\end{aligned}$$

Figure 6. Mobile Evaluation Contexts

$$\psi, \mathbf{ldc.i4}\ i4 \leadsto \psi, \boxed{i4} \tag{1}$$

$$\frac{\psi, I \leadsto \psi', I'}{\psi, E[I] \leadsto \psi', E[I']} \tag{2}$$

$$\frac{\text{if } i4 = 0 \text{ then } j = 3 \text{ else } j = 2}{\psi, \boxed{i4}\ I_2\ I_3\ \mathbf{cond} \leadsto \psi, I_j} \tag{3}$$

$$\psi, I_1\ I_2\ \mathbf{while} \leadsto \psi, I_1\ (I_2; (I_1\ I_2\ \mathbf{while}))\ \boxed{\mathbf{0}}\ \mathbf{cond} \tag{4}$$

$$\psi, \boxed{v}; I_2 \leadsto \psi, I_2 \tag{5}$$

$$\frac{0 \le j \le n}{(h, s(v_0, \ldots, v_n)), \mathbf{ldarg}\ j \leadsto (h, s(v_0, \ldots, v_n)), \boxed{v_j}} \tag{6}$$

$$\frac{0 \le j \le n}{\begin{array}{l}(h, s(v_0, \ldots, v_n)), \boxed{v}\ \mathbf{starg}\ j \leadsto \\ \quad (h, s(v_0, \ldots, v_{j-1}, v, v_{j+1}, \ldots, v_n)), \boxed{\mathbf{0}}\end{array}} \tag{7}$$

$$\frac{\ell \notin \mathit{Dom}(h) \qquad n = \mathit{fields}(C)}{\begin{array}{l}(h, s), \boxed{v_1}\ \ldots\ \boxed{v_n}\ \mathbf{newobj}\ C(\mu_1, \ldots, \mu_n) \leadsto \\ \quad (h[\ell \mapsto \mathit{obj}_C\{f_i = v_i \mid i \in 1..n\}^{\epsilon}], s), \boxed{\ell}\end{array}} \tag{8}$$

$$\frac{\mathit{methodbody}(C{::}m.\mathit{Sig}) = I}{(h, s), \boxed{v_0}\ \ldots\ \boxed{v_n}\ \mathbf{callvirt}\ C{::}m.\mathit{Sig} \leadsto (h, s(v_0, \ldots, v_n)), I\ \mathbf{ret}} \tag{9}$$

$$(h, sa), \boxed{v}\ \mathbf{ret} \leadsto (h, s), \boxed{v} \tag{10}$$

$$\frac{h(\ell) = \mathit{obj}_C\{\ldots, f = v, \ldots\}^{\overrightarrow{e}}}{(h, s), \boxed{\ell}\ \mathbf{ldfld}\ \mu\ C{::}f \leadsto (h, s), \boxed{v}} \tag{11}$$

$$\frac{h(\ell) = \mathit{obj}_C\{\ldots, f = v, \ldots\}^{\overrightarrow{e}}}{(h, s), \boxed{\ell}\ \boxed{v'}\ \mathbf{stfld}\ \mu\ C{::}f \leadsto (h[\ell \mapsto \mathit{obj}_C[f \mapsto v']], s), \boxed{\mathbf{0}}} \tag{12}$$

$$\frac{h(\ell) = \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}}}{(h, s), \boxed{\ell}\ \mathbf{evt}\ e_1 \leadsto (h[\ell \mapsto \mathit{obj}_C\{\ldots\}^{\overrightarrow{e}e_1}], s), \boxed{\mathbf{0}}} \tag{13}$$

$$\frac{\ell \notin \mathit{Dom}(h)}{(h, s), \mathbf{newpackage}\ C \leadsto (h[\ell \mapsto \mathit{pkg}(\cdot)], s), \boxed{\ell}} \tag{14}$$

$$\frac{h(\ell) = \mathit{pkg}(\ldots)}{(h, s), \boxed{\ell}\ \boxed{\ell'}\ \boxed{\mathit{rep}_C(H)}\ \mathbf{pack} \leadsto (h[\ell \mapsto \mathit{pkg}(\ell', \mathit{rep}_C(H))], s), \boxed{\mathbf{0}}} \tag{15}$$

$$\frac{h(\ell) = \mathit{pkg}(\ell', \mathit{rep}_C(H)) \qquad 0 \le j \le n}{\begin{array}{l}(h, s(v_0, \ldots, v_n)), \boxed{\ell}\ \mathbf{unpack}\ j \leadsto \\ \quad \big(h[\ell \mapsto \mathit{pkg}(\cdot)], s(v_0, \ldots, v_{j-1}, \mathit{rep}_C(H), v_{j+1}, \ldots, v_n)\big), \boxed{\ell'}\end{array}} \tag{16}$$

$$\frac{\text{if } \mathrm{test}_{C,k}(\mathit{rep}_C(H)) = 0 \text{ then } j = 3 \text{ else } j = 2}{\psi, \boxed{\mathit{rep}_C(H)}\ I_2\ I_3\ \mathbf{condst}\ C, k \leadsto \psi, I_j} \tag{17}$$

$$\frac{\mathit{arity}(\mathrm{hc}_{C,k}) = n}{\psi, \boxed{v_1}\ \ldots\ \boxed{v_n}\ \mathbf{newhist}\ C, k \leadsto \psi, \boxed{\mathrm{hc}_{C,k}(v_1, \ldots, v_n)}} \tag{18}$$

Figure 7. Small-step Operational Sematics for Mobile

```
1 (newobj C()) starg 1;
2 (ldarg 1) evt e1;
3 (ldarg 1) evt e2;
4 (newpackage C) starg 2;
5 (ldarg 2) (ldarg 1) (newhist C,0) pack;
6 (...) (ldarg 2) stfld ...;
7 ((ldarg 2) unpack 4) starg 3;
8 (ldarg 3) ((ldarg 4) evt e1) (...) condst C,0
```

Figure 8. Sample Mobile program

The operational semantics given in Figure 7 are for a single-threaded virtual machine without support for finalizers. To model concurrency, one could extend our stacks to consist of multiple threads and add a small-step rule that non-deterministically chooses which thread to execute next. Finalizers could be modeled by adding another small-step rule that non-deterministically forks a finalizer thread whenever an object is unreachable. Our implementation supports concurrency and finalizers, but to simplify the presentation, we leave the analysis of these language features to future work.

4.3 Type System

Mobile's type system considers each Mobile term to be a linear operator from a history map and frame list (describing the initial heap and stack, respectively) to a new history map and frame list (describing the heap and stack yielded by the operation) along with a return type. That is, we write $\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau')$ if term I, when evaluated in typing context Γ, takes history map Ψ and frame list $\overrightarrow{Fr}$ (in which any typing variables are bound in context Γ) to new history map Ψ' and new frame list $\overrightarrow{Fr}'$, and yields a value of type τ' (if it terminates). Any new typing variables appearing in $\overrightarrow{Fr}'$ and τ' are bound in context Γ'. A method signature (see Figure 3) is the type assigned to the term comprising its body.

Below, we provide an informal description of Mobile's typing rules by walking the type-checking algorithm through the sample Mobile program given in Figure 8. A complete list of typing rules is stated formally in the appendix.

Line 1 of the sample program creates a new object of class C and stores it in local register 1. When a new security-relevant object is created, Mobile's type system assigns it a fresh object identity variable ℓ. The return type of the newly created object is thus $C\langle\ell\rangle$ and the new history map yielded by the operation satisfies $\Psi'(\ell) = \epsilon$; that is, new objects are initially assigned the empty trace.

As security-relevant events are performed on the object, the type system tracks these changes by statically updating its history map to append these new events to the sequence it recorded in its history map. So for example, after processing lines 2–3 of the sample program, which perform events e_1 and e_2 on the object in local register 1, the type-checker's new history map would satisfy $\Psi'(\ell) = e_1 e_2$. At each point that a security-relevant event is performed, the type system ensures that the new trace satisfies a prefix of the security policy. For example, when type-checking line 3, the type-checker would verify that $e_1 e_2 \subseteq pre(policy(C))$, where $policy(C)$ denotes the set of acceptable traces assigned by the security policy to class C, and $pre(policy(C))$ denotes the set of prefixes of members of set $policy(C)$.

Security-relevant objects of type $C\langle\ell\rangle$ are like typical objects except that they are not permitted to escape to the heap. That is, they cannot be assigned to object fields. In order to leak a security-relevant object to the heap, a Mobile program must first store it in a package using a **pack** instruction. This requires three steps: (1) A package must be created via a **newpackage** instruction. (2) A runtime state value must be created that accurately reflects the state of the object to be packed. This is accomplished via the **newhist** instruction, which is described in more detail below. (3) Finally, the **pack** operation is used to store the object and the runtime state value into the package. Lines 4 and 5 of the sample program illustrate these three steps. Line 4 creates a new package and stores it in local register 2. Line 5 then fills the package using the object in local register 1 along with a newly created runtime state value.

In order for Mobile's type system to accept a **pack** operation, it must be able to statically verify that the runtime state value is an accurate abstraction of the object being packed. That is, if the runtime state value has type $Rep_C\langle H\rangle$, then the type system requires that $\Psi(\ell) \subseteq H$ where ℓ is the object identity variable of the object being packed. Additionally, since packed objects are untracked and therefore might continue to exist until the program terminates, packed objects must satisfy the security policy. That is, we require that $\Psi(\ell) \subseteq policy(C)$.

Packages that contain security-relevant objects can leak to the heap, as illustrated by line 6 of the sample program, which stores the package to a field of some other object. Since only packed objects can leak to the heap, the restriction that packed objects must be in a policy-adherent state is a potential limitation of the type system. That is, it might often be desirable to leak an object that is not yet in a policy-adherent state to the heap, but later retrieve it and restore it to a policy-adherent state before the program terminates. In §5 we show how Mobile implementations can use finalizer code to avoid this restriction and leak objects to the heap even when they are not yet in a policy-adherent state.

After a **pack** operation, the type system removes object identity variable ℓ from the history map. Hence, after line 5 of the sample program, $\Psi'(\ell)$ is undefined and the object that was packed becomes inaccessible. If the program were to subsequently attempt to load from local register 1 (before replacing its contents with something else), the type-checker would reject the code because that register now contains a value with an invalid type. Object identity variable ℓ can therefore be thought of as a capability that has been revoked from the local scope and given to the package.

In order to perform more security-relevant events on an object, a Mobile program must first reacquire a capability for the object by unpacking the object from its package via an **unpack** instruction. Line 7 of the sample program unpacks the package in local register 2, storing the extracted object in local register 3 and storing the runtime state value that was packaged with it in local register 4. Since packages and the objects they contain are not tracked by the type system, the type system cannot statically determine the history of a freshly unpacked object. All that is statically known is that the runtime state value that will be yielded at runtime by the **unpack** instruction will be an accurate representation of the unpacked object's history. To reflect this information statically, the type system assigns a fresh object identity variable ℓ' to the unpacked object and a fresh history variable θ to the unknown history. The unpacked object and runtime state value then have types $C\langle\ell'\rangle$ and $Rep_C\langle\theta\rangle$, respectively, and the new history map satisfies $\Psi'(\ell') = \theta$. The type $C\langle ? \rangle$ of a package can hence be thought of as an existential type binding type variables ℓ' and θ.

If the sample program were at this point to perform security-relevant event e on the newly unpacked object, Mobile's type system would reject because it would be unable to statically verify that $\theta e \subseteq policy(C)$ (since nothing is statically known about history θ). However, a Mobile program can perform additional **evt** operations on the object by first dynamically testing the runtime state value yielded by the **unpack** operation. If a Mobile program dynamically tests a value of type $Rep_C\langle\theta\rangle$, Mobile's type system can statically infer information about history θ within the branches of the conditional. For example, if a **condst** instruction is used to test a value with type $Rep_C\langle\theta\rangle$ for equality with a value of type

$Rep_C\langle e_1e_2\rangle$, then in the positive branch of the conditional, the type system can statically infer that $\theta = e_1e_2$. If $policy(C) = (e_1e_2)^\omega$, then a Mobile program could execute I $\mathbf{evt}$ e_1 within the positive branch of such a conditional (where I is the object that was unpacked), because $e_1e_2e_1 \subseteq pre((e_1e_2)^\omega)$; but the type-checker would reject a program that executed I $\mathbf{evt}$ e_2 in the positive branch, since $e_1e_2e_2 \not\subseteq pre((e_1e_2)^\omega)$.

Mobile supports many possible schemes for representing histories at runtime and for testing them, so rather than fixing particular operations for constructing runtime state values and particular operations for testing them, we instead assume only that there exists a countable collection of constructors $\mathbf{newhist}$ C, k and conditionals $\mathbf{condst}$ C, k for all integers k, that construct runtime state values and test runtime state values (respectively) for objects of class C. We then abstractly define $HC_{C,k}(\ldots)$ to be the type $Rep_C\langle H\rangle$ of a history value constructed using constructor k for security-relevant class C, and we define $ctx^+_{C,k}(H,\Psi)$ and $ctx^-_{C,k}(H,\Psi)$ to be the object history maps that refine Ψ in the positive and negative branches (respectively) of a conditional that performs test k on a history value of type $Rep_C\langle H\rangle$. Mobile supports any such refinement that is sound in the sense that

$$\mathrm{test}_{C,k}(H) = 0 \Longrightarrow \Psi \preceq ctx^-_{C,k}(H,\Psi)(\ell) \qquad (19)$$

and

$$\mathrm{test}_{C,k}(H) \neq 0 \Longrightarrow \Psi \preceq ctx^+_{C,k}(H,\Psi)(\ell) \qquad (20)$$

We further assume that each history type constructor $HC_{C,k}(\ldots)$ accurately reflects its runtime implementation, in the sense that for all history value types $Rep_{C_1}\langle H_1\rangle, \ldots, Rep_{C_n}\langle H_n\rangle$ such that $n = arity(HC_{C,k})$, there exists some H such that

$$HC_{C,k}(Rep_{C_1}\langle H_1\rangle, \ldots, Rep_{C_n}\langle H_n\rangle) = Rep_C\langle H\rangle \qquad (21)$$

and

$$\mathrm{hc}_{C,k}(rep_{C_1}(H_1), \ldots, rep_{C_n}(H_n)) = rep_C(H) \qquad (22)$$

In the sample program, suppose that history value constructor $\mathbf{newhist}$ $C, 0$ takes no arguments and yields a runtime value that represents history e_1e_2; and suppose that conditional test $\mathbf{condst}$ $C, 0$ compares a runtime state value to the value that represents history e_1e_2. Formally, suppose that $HC_{C,0}() = Rep_C\langle e_1e_2\rangle$ and $ctx^+_{C,0}(\theta,\Psi) = \Psi[\theta \mapsto e_1e_2]$. Thus, in the positive branch of such a test, the type-checker's object history map can be refined by substituting e_1e_2 for any instances of the history variable being tested. Then if $policy(C) = (e_1e_2)^\omega$, a Mobile type-checker would accept the sample program. In the positive branch of the conditional in line 8, the type-checker would infer that the object in local register 4 has history e_1e_2, and therefore it is safe to perform event e_1 on it. However, if $policy(C) = e_1e_2e_2$, then the type-checker would reject, because $e_1e_2e_1$ is not a prefix of $e_1e_2e_2$.

In the negative branch of this conditional the type-checker can infer that the object in local register 4 has a history represented by a state value other than the one that it was tested against. The history map could therefore be refined by substituting history variable θ with the union of all of the history abstractions associated with all of the other possible runtime state values defined for that object's class.

Our implementation of Mobile implements history abstraction values as integers. Thus, it provides 2^{32} $\mathbf{newhist}$ operations for each security-relevant class C, defining $\mathrm{hc}_{C,k}() = k$ for all $k \in 0..2^{32}-1$. Tests $\mathbf{condst}$ of runtime state values are implemented as equality comparisons between the integer runtime state value to be tested and an integer constant. Thus, we define

$$\mathrm{test}_{C,k}(rep_C(\theta)) = \begin{cases} 1 & \text{if } rep_C(\theta) = k \\ 0 & \text{otherwise} \end{cases}$$
$$ctx^+_{C,k}(\theta,\Psi) = \Psi[\theta \mapsto \theta \cap H_k]$$
$$ctx^-_{C,k}(\theta,\Psi) = \Psi[\theta \mapsto \theta \cap (\cup_{i\neq k} H_i)]$$

for each integer $k \in 0..2^{32}-1$, where H_k is a closed history abstraction statically assigned to integer constant k. The assignments of closed history abstractions H_k to integers k are not trusted, so this mapping can be defined by the Mobile program itself (e.g., in settings where self-monitoring programs are produced by a common rewriter or where separately produced programs do not exchange objects) or by the policy-writer (in settings where the mapping must be defined at a system global level for consistency).

The above scheme allows a Mobile program to represent object security states at runtime with a security automaton of 2^{32} states or less. Each state of the automaton is assigned an integer constant k, and history abstraction H_k would denote the set of traces that cause the automaton to arrive in state k.

4.4 Policy Adherence of Mobile Programs

The operational semantics of Mobile presented in §4.2 permit untyped Mobile programs to enter bad terminal states—states in which the Mobile program has not been reduced to a value but no progress can be made. For example, an untyped Mobile program might attempt to load from a non-existent field or attempt to unpack an empty package (in which case no small-step rule can be applied). Mobile's type system presented in §4.3 prevents both policy violations and bad terminal states, except that it does not prevent $\mathbf{unpack}$ operations from being performed on empty packages. This reflects the reality that in practical settings there will always be bad terminal states that are not statically preventable. We prove below that Mobile programs well-typed with respect to a security policy will not violate the security policy when executed even if they enter a bad state.

Formally, we define well-typed by

Definition 1. A method $C{::}m.Sig$ with $Sig = \forall\Gamma_{in}.(\Psi_{in}, Fr_{in}) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau)$ is *well-typed* if and only if there exists a derivation for the typing judgment $\Gamma_{in} \vdash I : (\Psi_{in}, Fr_{in}) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau)$ where $I = methodbody(C{::}m.Sig)$.

Definition 2. A Mobile program is *well-typed* if and only if (1) for all $C{::}m.Sig \in Dom(methodbody)$, method $C{::}m.Sig$ is well-typed, and (2) there exists a method $C_{main}{::}main.Sig_{main} \in Dom(methodbody)$ with $Sig_{main} = \forall\Gamma_{in}.(\Psi_{in}, (\tau_1,\ldots,\tau_n)) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau_{out})$ such that for all substitutions $\sigma : \theta \to \overrightarrow{e}$ and all object identity variables $\ell{:}C \in (\Gamma_{in}, \Gamma_{out})$, if $\Psi_{out}(\ell) = H$ then $\sigma(H) \subseteq policy(C)$.

Part 2 of definition 2 captures the requirement that a Mobile program's entry method must have a signature that complies with the security policy on exit.

Policy violations are defined differently depending on whether the program terminates normally. If the program terminates normally, Mobile's type system guarantees that the resulting heap will be policy-adherent; whereas if the program does not terminate or enters a bad state, Mobile guarantees only that the heap at each evaluation step will be prefix-adherent, where policy- and prefix-adherence are defined as follows:

Definition 3 (Policy Adherent)**.** A heap h is *policy-adherent* if, for all class objects $obj_C\{\ldots\}^{\overrightarrow{e}} \in Rng(h)$, $\overrightarrow{e} \subseteq policy(C)$.

Definition 4 (Prefix Adherent)**.** A heap h is *prefix-adherent* if, for all class objects $obj_C\{\ldots\}^{\overrightarrow{e}} \in Rng(h)$, $\overrightarrow{e} \subseteq pre(policy(C))$.

To formalize the theorem, we first define a notion of consistency between a static typing context and a runtime memory state. We say that a memory store ψ *respects* an object identity context Ψ and a list of frames $\overrightarrow{Fr}$, written $\Gamma \vdash \psi : (\Psi; \overrightarrow{Fr})$ if all object fields and stack slots in ψ have values of appropriate types, and the heap in ψ is prefix-adherent. (See [17] for a formal definition.) The following two theorems then establish that well-typed Mobile programs do not violate the security policy.

Theorem 1 (Terminating Policy Adherence). *Assume that a Mobile program is well-typed, and that, as per Definition 2, its* main *method has signature* $Sig_{main} = \forall\Gamma_{in}.(\Psi_{in}, (\tau_1, \ldots, \tau_n)) \multimap \exists\Gamma_{out}.(\Psi_{out}, Fr_{out}, \tau_{out})$*. If* $\Gamma_{in} \vdash \psi : (\Psi_{in}; Fr)$ *holds and if* $\psi, methodbody(C_{main}::main.Sig) \rightsquigarrow^* (h', s'), \boxed{v}$ *holds, then* h' *is policy-adherent.*

Proof. Omitted for brevity. See [17]. □

Theorem 2 (Non-terminating Prefix Adherence). *Assume that a Mobile program is well-typed, and assume that* $\Gamma \vdash I : (\Psi; \overrightarrow{Fr}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{Fr}'; \tau)$ *and* $\Gamma \vdash (h; s) : (\Psi; \overrightarrow{Fr})$ *hold. If* h *is prefix-adherent and* $(h, s), I \rightsquigarrow^n (h', s'), I'$ *holds, then* h' *is prefix-adherent.*

Proof. Omitted for brevity. See [17]. □

An important consequence of both of these theorems is that Mobile can be implemented on existing .NET systems without modifying the memory model to store object traces at runtime. Since a static type-checker can verify that Mobile code is well-typed, and since well-typed code never exhibits a trace that violates the security policy, the runtime system need not store or monitor object traces to prevent security violations.

5. Implementation

Our prototype implementation of Mobile consists of a type-checker for Mobile's type system extended to the full managed subset of Microsoft's .NET CIL (minus reflection). The type-checker was written in Ocaml (about one thousand lines of code) and uses Microsoft's .NET ILX SDK [31] to read and manipulate .NET bytecode binaries. Mobile programs are .NET CIL programs with typing annotations encoded as .NET method attributes. The Mobile type-checker reads these (untrusted) annotations and verifies them in the course of type-checking.

Our implementation allows security policies to identify method calls as security-relevant events. Thus, security policies can constrain the usage of resources provided by the CLR by monitoring CLR method calls and the objects they return. Our type-checker can, in principle, regard any CIL instruction as a security-relevant event, but we leave practical investigation of this feature to future work.

Operations **pack** and **unpack** are implemented as method calls to the (very small) trusted C# library given in Figure 9. Observe that C#'s `lock` construct is used to make both operations atomic. History abstraction values are implemented as integers. Thus, our **newhist** operation is simply a **ldc**.**i4** instruction that loads an integer constant onto the evaluation stack. Policies can statically declare for each integer constant a closed history abstraction that integer represents when used as a runtime state value. Tests of runtime state values consist of equality comparisons with integer constants in the manner described in §4.3. As described in §4.3, this implementation suffices to support IRM's that model security policies as finite-state security automata.

The type-checker must verify subset relations over the language of history abstractions given in Figure 3. Although deciding subset for ω-regular expressions with variables and intersection is not tractable in general, the task is simplified by observing that real Mobile code only introduces history variables at the beginnings of expressions (when an object is unpacked) and only introduces intersections that involve a variable and a closed history abstraction (when a runtime state value is tested). The resulting sub-language can be decided using a simple regular expression subset algorithm (proof omitted for brevity).

```
class Package {
   private object obj;
   private int state;

   public void Pack(object o, int s) {
      lock (this) { obj=o; state=s; }
   }

   public object Unpack(ref int s) {
      lock (this) {
         object o=obj;
         if (o==null) throw new EmptyPackage();
         obj=null; s=state;
         return o;
      }
   }
}
```

Figure 9. Implementation of **pack** and **unpack**

Our type-checker also recognizes method annotations attached to finalizers of security-relevant classes. A finalizer's precondition must be satisfied whenever an object of its class escapes to the heap (i.e., when it is packed), since at any point after that, its package object could become orphaned and then garbage-collected. By the time a program terminates, all of its objects are guaranteed to satisfy their finalizers' postconditions, since at that point any remaining objects will be garbage-collected. This allows an IRM to leak security-relevant objects to the heap (in packages) even when they are not yet in a policy-adherent state, as long as the object's finalizer suffices to restore it to a policy-adherent state once garbage-collection occurs.

To test our implementation, we wrote a simple rewriter like that described in §3, and used it to enforce a security policy that allows each .NET network socket object to accept at most n connections during the program's lifetime (where n is a parameter specified by the policy-writer). Such a policy might be used, for example, to force applications to relinquish control of network ports after a certain amount of activity. We applied this policy to a small multithreaded webserver written in C#. The original application binary was 20K in size and rewriting did not alter its size. (Padding introduced by the CLI binary format masked the small overhead introduced by additional instructions and annotations.) Hand-counting the material inserted by the rewriter revealed approximately 83 bytes in additional instructions and 117 bytes in annotations. Rewriting took 0.12 seconds and type-checking took 0.09 seconds on a 1.8GHz Pentium. We benchmarked both the original and rewritten webservers by using WebStone to simulate two clients retrieving five webpages ranging in size from 500 bytes to 5 megabytes. WebStone reported that the rewritten webserver exhibited an average throughput rate that was 99.97% of the original webserver's.

The aforementioned test is obviously not a definitive evaluation of the feasibility of automated rewriting; much work remains to be done in terms of evaluating the approach on richer policies and applications. However, we consider it to be preliminary evidence that rewriting can be automated in a way that produces acceptable annotations.

6. Conclusions and Future Work

Mobile's type system and the theorems presented in §4.4 show that a common style of IRM, in which extra state variables and guards that model a security automaton have been in-lined into the untrusted code, can be independently verified by a type-checker, eliminating the need to trust the rewriter that produced the IRM. We verify policies that are universally quantified over unbounded collections of objects—that is, policies that require each object to exhibit a history of security-relevant events that conforms to some stated property. The language of security policies is left abstract and could consist of DFA's, LTL expressions, or any computable language of finite and infinite event sequences.

Our implementation of Mobile for managed Microsoft .NET CIL expresses security policies as ω-regular expressions. We verify such policies in the presence of exceptions, concurrency, finalizers, and non-termination, demonstrating that Mobile can be scaled to real type-safe, low-level languages.

Our presentation of Mobile has not addressed issues of object inheritance of security-relevant classes. Future work should examine how to safely express and implement policies that require objects related by inheritance to conform to different properties. A type-checker for such a system would need to identify when a typecast at runtime could potentially lead to a violation of the policy and provide a means for policy-adherent programs to perform necessary typecasts.

Another open problem is how to support a wider range of IRM implementations. Mobile supports only a specific (but typical) treatment of runtime state, wherein each security-relevant object is paired with a dynamic representation of its state every time it is leaked to the heap. In some settings, it may be desirable to implement IRM's that store an object's dynamic state differently, such as in a separate array rather than packaged together with the object it models. Type systems for coordinated data structures [27] could potentially be leveraged to enforce invariants over these decoupled objects and states.

We chose a type system for Mobile that statically tracks control flow in a data-insensitive manner, with ω-regular expressions denoting sets of event sequences. This approach is appealing because there is a natural rewriting strategy (outlined in §3) whereby well-typed Mobile code can be automatically generated from untrusted CIL code. A more powerful type system could employ a richer language like Hoare Logic [19] to track data-sensitive control flow. This could allow clever rewriters to eliminate additional runtime checks by statically proving that they are unnecessary. However, formulating a sound and complete Hoare Logic for .NET that includes objects and concurrency is challenging; furthermore, the burden of producing useful proofs in this logic would be pushed to the rewriter. Future work should investigate rewriting strategies that could make such an approach worthwhile.

Finally, not every enforceable security policy can be couched as a computable property that is universally quantified over object instances. For example, one potentially useful policy is one that requires that for every file object opened for writing, there exists an encryptor object to which its output stream has been linked. Such a policy is not supported by Mobile because it regards both universal and existentially quantified properties that relate multiple object instances. Future work should consider how to implement IRM's that enforce such policies, and how these implementations could be type-checked so as to statically verify that the IRM satisfies the security policy.

Acknowledgments

The authors are indebted to Matthew Fluet, Michael Hicks, and Amal Ahmed for their helpful critiques of this paper.

References

[1] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314, Chicago, Illinois, June 2005.

[2] Andrew Bernard. *Engineering Formal Security Policies for Proof-Carrying Code*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 2004.

[3] Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. In *18th International Conference on Automated Deduction*, pages 31–46, Copenhagen, Denmark, July 2002.

[4] Feng Chen and Grigore Rosu. Java-MOP: A monitoring oriented programming environment for Java. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550, Edinburgh, U.K., April 2005.

[5] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310, Seattle, Washington, November 2002.

[6] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *15th European Conference for Object-Oriented Programming (ECOOP)*, pages 53–76, June 2001.

[7] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, Snowbird, Utah, June 2001.

[8] Robert DeLine and Manuel Fähndrich. The Fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, Redmond, Washington, January 2004.

[9] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *18th European Conference on Object-Oriented Programming*, pages 465–490, Oslo, Norway, June 2004.

[10] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002.

[11] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.

[12] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

[13] David Evans and Andrew Twynman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, Oakland, California, May 1999.

[14] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Berlin, Germany, June 2002.

[15] Li Gong. Java^{TM} 2 platform security architecture, version 1.2. Whitepaper. © 1997–2002 Sun Microsystems, Inc.

[16] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *28th ACM Symposium on Principles of Programming Languages*, pages 248–260, London, United Kingdom, January 2001.

[17] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .NET. Technical Report TR-2005-2003, Cornell University, Ithaca, New York, November 2005.

[18] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions On Programming Languages And Systems (TOPLAS)*, 28(1):175–205, January 2006.

[19] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[20] Andrew Kennedy and Don Syme. The design and implementation of generics for the .NET common language runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Snowbird, Utah, June 2001.

[21] Moonjoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg V. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, March 2004.

[22] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[23] Greg Morrisett, Amal Ahmed, and Matthew Fluet. L^3: A linear language with locations. In *7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, pages 293–307, Nara, Japan, April 2005.

[24] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999.

[25] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[26] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *15th Annual Conference of the European Association for Computer Science Logic, LNCS*, pages 1–19, Paris, France, 2001. Springer-Verlag.

[27] Michael F. Ringenburg and Dan Grossman. Types for describing coordinated data structures. In *ACM SIGPLAN International Workshop on Types in Languages, Design and Implementation (TLDI)*, pages 25–36, Long Beach, California, January 2005.

[28] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.

[29] Christian Skalka and Scott F. Smith. History effects and verification. In *Asian Programming Languages Symposium (APLAS)*, pages 107–128, November 2004.

[30] Frederick Smith, David Walker, and Greg Morrisett. Alias types. *Lecture Notes in Computer Science*, 1782:366–381, March 2000.

[31] Don Syme. ILX: Extending the .NET Common IL for functional language interoperability. In Nick Benton and Andrew Kennedy, editors, *First International Workshop on Multi-Language Infrastructure and Interoperability*, volume 59.1, Florence, Italy, September 2001.

[32] David Walker. A type system for expressive security policies. In *27th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 254–267, January 2000.

Appendix

The following is a formal statement of Mobile's typing rules.

$$\frac{}{\Gamma \vdash \mathbf{ldc.i4}\, n : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap (\Psi; \overrightarrow{\mathit{Fr}}; \mathbf{int32})} \tag{23}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{\mathit{Fr}}_1; \mathbf{int32}) \\ \Gamma, \Gamma_1 \vdash I_i : (\Psi_1; \overrightarrow{\mathit{Fr}}_1) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'; \tau)\ \forall i \in \{2,3\}\end{array}}{\Gamma \vdash I_1\ I_2\ I_3\ \mathbf{cond} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1, \Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'; \tau)} \tag{24}$$

$$\frac{\Gamma, \Gamma' \vdash I_1\ I_2\ \boxed{\mathbf{0}}\ \mathbf{cond} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi; \overrightarrow{\mathit{Fr}}; \mathbf{void})}{\Gamma, \Gamma' \vdash I_1\ I_2\ \mathbf{while} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi; \overrightarrow{\mathit{Fr}}; \mathbf{void})} \tag{25}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{\mathit{Fr}}_1; \mathbf{void}) \\ \Gamma, \Gamma_1 \vdash I_2 : (\Psi_1; \overrightarrow{\mathit{Fr}}_1) \multimap \exists\Gamma_2.(\Psi'; \overrightarrow{\mathit{Fr}}'; \tau)\end{array}}{\Gamma \vdash I_1; I_2 : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1, \Gamma_2.(\Psi'; \overrightarrow{\mathit{Fr}}'; \tau)} \tag{26}$$

$$\frac{\begin{array}{c}\ell \in \mathit{Dom}(\Psi') \qquad \mathit{field}(C,f) = \mu \\ \Gamma \vdash I : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'; C\langle\ell\rangle)\end{array}}{\Gamma \vdash I\ \mathbf{ldfld}\, \mu\, C{::}f : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'; \mu)} \tag{27}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{\mathit{Fr}}_1; C\langle\ell\rangle) \qquad \ell \in \mathit{Dom}(\Psi') \\ \Gamma, \Gamma_1 \vdash I_2 : (\Psi_1; \overrightarrow{\mathit{Fr}}_1) \multimap \exists\Gamma_2.(\Psi'; \overrightarrow{\mathit{Fr}}'; \mu) \quad \mathit{field}(C,f) = \mu\end{array}}{\Gamma \vdash I_1\ I_2\ \mathbf{stfld}\, \mu\, C{::}f : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1, \Gamma_2.(\Psi'; \overrightarrow{\mathit{Fr}}'; \mathbf{void})} \tag{28}$$

$$\frac{0 \le j \le n}{\Gamma \vdash \mathbf{ldarg}\, j : (\Psi; \overrightarrow{\mathit{Fr}}(\tau_0, \ldots, \tau_n)) \multimap (\Psi; \overrightarrow{\mathit{Fr}}(\tau_0, \ldots, \tau_n); \tau_j)} \tag{29}$$

$$\frac{\Gamma \vdash I : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'(\tau_0, \ldots, \tau_n); \tau) \qquad 0 \le j \le n}{\begin{array}{l}\Gamma \vdash I\ \mathbf{starg}\, j : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \\ \quad \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'(\tau_0, \ldots, \tau_{j-1}, \tau, \tau_{j+1}, \ldots, \tau_n); \mathbf{void})\end{array}} \tag{30}$$

$$\frac{\begin{array}{c}\Gamma, \Gamma_1, \ldots, \Gamma_{i-1} \vdash I_i : (\Psi_{i-1}; \overrightarrow{\mathit{Fr}}_{i-1}) \multimap \\ \qquad \exists\Gamma_i.(\Psi_i; \overrightarrow{\mathit{Fr}}_i; \mu_i) \quad \forall i \in 1..n \\ n = \mathit{fields}(C) \quad \ell \notin \mathit{Dom}(\Gamma, \Gamma_1, \ldots, \Gamma_n) \quad \epsilon \in \mathit{pre}(\mathit{policy}(C))\end{array}}{\begin{array}{l}\Gamma \vdash I_1\ \ldots\ I_n\ \mathbf{newobj}\, C(\mu_1, \ldots, \mu_n) : \\ \quad (\Psi_0; \overrightarrow{\mathit{Fr}}_0) \multimap \exists\Gamma_1, \ldots, \Gamma_n, \ell{:}C.(\Psi_n \star (\ell \mapsto \epsilon); \overrightarrow{\mathit{Fr}}_n; C\langle\ell\rangle)\end{array}} \tag{31}$$

$$\frac{\begin{array}{c}\Gamma_0, \ldots, \Gamma_j \vdash I_j : (\Psi_j, \overrightarrow{\mathit{Fr}}_j) \multimap \exists\Gamma_{j+1}.(\Psi_{j+1}, \overrightarrow{\mathit{Fr}}_{j+1}, \tau_j)\ \forall j \in 0..n \\ \tau_0 = C\langle\ell\rangle \quad \ell \in \mathit{Dom}(\Psi_{n+1}) \quad C{::}m.\mathit{Sig} \in \mathit{Dom}(\mathit{methodbody}) \\ \Gamma_0, \ldots, \Gamma_n \vdash \mathit{Sig} <: (\Psi_{in}, (\tau_0, \ldots, \tau_n)) \multimap \exists\Gamma_{out}.(\Psi_{out}, \mathit{Fr}_{out}, \tau) \\ \Psi_{n+1} = \Psi_{unused} \star \Psi_{in}\end{array}}{\begin{array}{l}\Gamma_0 \vdash I_0\ \ldots\ I_n\ \mathbf{callvirt}\, C{::}m.\mathit{Sig} : \\ \quad (\Psi_0, \overrightarrow{\mathit{Fr}}_0) \multimap \exists\Gamma_1, \ldots, \Gamma_{n+1}, \Gamma_{out}.(\Psi_{unused} \star \Psi_{out}, \overrightarrow{\mathit{Fr}}_{n+1}, \tau)\end{array}} \tag{32}$$

$$\frac{\begin{array}{c}He \subseteq \mathit{pre}(\mathit{policy}(C)) \\ \Gamma \vdash I : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi' \star (\ell \mapsto H); \overrightarrow{\mathit{Fr}}'; C\langle\ell\rangle)\end{array}}{\Gamma \vdash I\ \mathbf{evt}\, e : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi' \star (\ell \mapsto He); \overrightarrow{\mathit{Fr}}'; \mathbf{void})} \tag{33}$$

$$\frac{\ell \notin \mathit{Dom}(\Gamma)}{\Gamma \vdash \mathbf{newpackage}\, C : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\ell{:}C\langle ? \rangle.(\Psi; \overrightarrow{\mathit{Fr}}; C\langle ? \rangle)} \tag{34}$$

$$\frac{\begin{array}{c}H \subseteq H' \subseteq \mathit{policy}(C) \\ \Gamma \vdash I_1 : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{\mathit{Fr}}_1; C\langle ? \rangle) \\ \Gamma, \Gamma_1 \vdash I_2 : (\Psi_1; \overrightarrow{\mathit{Fr}}_1) \multimap \exists\Gamma_2.(\Psi_2; \overrightarrow{\mathit{Fr}}_2; C\langle\ell\rangle) \\ \Gamma, \Gamma_1, \Gamma_2 \vdash I_3 : (\Psi_2; \overrightarrow{\mathit{Fr}}_2) \multimap \exists\Gamma_3.(\Psi' \star (\ell \mapsto H); \overrightarrow{\mathit{Fr}}'; \mathit{Rep}_C\langle H' \rangle)\end{array}}{\Gamma \vdash I_1\ I_2\ I_3\ \mathbf{pack} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1, \Gamma_2, \Gamma_3.(\Psi'; \overrightarrow{\mathit{Fr}}'; \mathbf{void})} \tag{35}$$

$$\frac{\begin{array}{c}\ell \notin \mathit{Dom}(\Psi') \qquad \theta \notin \mathit{Dom}(\Gamma) \\ \Gamma \vdash I : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'(\tau_0, \ldots, \tau_n); C\langle ? \rangle)\end{array}}{\begin{array}{l}\Gamma \vdash I\ \mathbf{unpack}\, j : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma', \ell{:}C, \theta. \\ \quad (\Psi', \ell \mapsto \theta; \overrightarrow{\mathit{Fr}}'(\tau_0, \ldots, \tau_{j-1}, \mathit{Rep}_C\langle\theta\rangle, \tau_{j+1}, \ldots, \tau_n); C\langle\ell\rangle)\end{array}} \tag{36}$$

$$\frac{\begin{array}{c}\Gamma \vdash I_1 : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1.(\Psi_1; \overrightarrow{\mathit{Fr}}_1; \mathit{Rep}_C\langle H \rangle) \\ \Gamma, \Gamma_1 \vdash I_2 : (\mathit{ctx}^{+}_{C,k}(H, \Psi_1); \overrightarrow{\mathit{Fr}}_1) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'; \tau) \\ \Gamma, \Gamma_1 \vdash I_3 : (\mathit{ctx}^{-}_{C,k}(H, \Psi_1); \overrightarrow{\mathit{Fr}}_1) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'; \tau)\end{array}}{\Gamma \vdash I_1\ I_2\ I_3\ \mathbf{condst}\, k : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma_1, \Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'; \tau)} \tag{37}$$

$$\frac{\Gamma \vdash I_i : (\Psi_{i-1}; \overrightarrow{\mathit{Fr}}_{i-1}) \multimap \exists\Gamma_i.(\Psi_i; \overrightarrow{\mathit{Fr}}_i; \mathit{Rep}_{C_i}\langle H_i \rangle)\ \forall i \in 1..n}{\begin{array}{l}\Gamma \vdash I_1\ \ldots\ I_n\ \mathbf{newhist}\, C, k : (\Psi_0; \overrightarrow{\mathit{Fr}}_0) \multimap \exists\Gamma_1, \ldots, \Gamma_n. \\ \qquad (\Psi_n; \overrightarrow{\mathit{Fr}}_n; HC_{C,k}(\mathit{Rep}_{C_1}\langle H_1 \rangle, \ldots, \mathit{Rep}_{C_n}\langle H_n \rangle))\end{array}} \tag{38}$$

$$\frac{\begin{array}{c}\Gamma_1, \Gamma' \vdash I : (\Psi_1; \overrightarrow{\mathit{Fr}}_1) \multimap \exists\Gamma_2.(\Psi_2; \overrightarrow{\mathit{Fr}}_2; \tau) \\ \Psi'_1 \preceq \Psi_1 \quad \overrightarrow{\mathit{Fr}}'_1 \preceq \overrightarrow{\mathit{Fr}}_1 \quad \Psi_2 \preceq \Psi'_2 \quad \overrightarrow{\mathit{Fr}}_2 \preceq \overrightarrow{\mathit{Fr}}'_2 \quad \tau \preceq \tau'\end{array}}{\Gamma_1, \Gamma' \vdash I : (\Psi'_1; \overrightarrow{\mathit{Fr}}'_1) \multimap \exists\Gamma_2, \Gamma'.(\Psi'_2; \overrightarrow{\mathit{Fr}}'_2; \tau')} \tag{39}$$

$$\frac{}{\Gamma \vdash \boxed{\mathbf{0}} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap (\Psi; \overrightarrow{\mathit{Fr}}; \mathbf{void})} \tag{40}$$

$$\frac{}{\Gamma \vdash \boxed{i4} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap (\Psi; \overrightarrow{\mathit{Fr}}; \mathbf{int32})} \tag{41}$$

$$\frac{\Psi = \Psi' \star (\ell \mapsto H)}{\Gamma, \ell{:}C \vdash \boxed{\ell} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap (\Psi; \overrightarrow{\mathit{Fr}}; C\langle\ell\rangle)} \tag{42}$$

$$\frac{}{\Gamma, \ell{:}C\langle ? \rangle \vdash \boxed{\ell} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap (\Psi; \overrightarrow{\mathit{Fr}}; C\langle ? \rangle)} \tag{43}$$

$$\frac{}{\Gamma \vdash \boxed{\mathit{rep}_C(H)} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap (\Psi; \overrightarrow{\mathit{Fr}}; \mathit{Rep}_C\langle H \rangle)} \tag{44}$$

$$\frac{\Gamma \vdash I : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'\, \mathit{Fr}_0; \tau)}{\Gamma \vdash I\ \mathbf{ret} : (\Psi; \overrightarrow{\mathit{Fr}}) \multimap \exists\Gamma'.(\Psi'; \overrightarrow{\mathit{Fr}}'; \tau)} \tag{45}$$

Judgment $\Gamma \vdash \mathit{Sig}_1 <: \mathit{Sig}_2$ in rule 32 asserts that Sig_1 alpha-varies to Sig_2.